$P4_{16}$ Portable Switch Architecture (PSA)

(draft)

The P4.org language consortium

May 15, 2017

Abstract

P4 is a language for expressing how packets are processed by the data plane of a programmable network forwarding element. P4 programs specify how the various programmable blocks of a target architecture are programmed and connected. The Portable Switch Architecture (PSA) is target architecture that describes common capabilities of network switch devices which process and forward packets across multiple interface ports.

Contents

1. Target Architecture Model

The Portable Switch Architecture (PSA) Model has six programmable P4 blocks and two fixed-function blocks, as shown in Figure 1. Programmable blocks are hardware blocks whose function can be programmed using the P4 language. The Packet buffer and Replication Engine (PRE) and the Buffer Queuing Engine (BQE) are target dependent functional blocks that may be configured for a fixed set of operations.

Incoming packets are parsed and have their checksums validated and are then passed to an ingress match action pipeline, which makes decisions on where the packets should go. After the ingress pipeline, the packet may be buffered and/or replicated (sent to multiple egress ports). For each such egress port, the packet passes through an egress match action pipeline and a checksum update calculation before it is deparsed and queued to leave the pipeline..

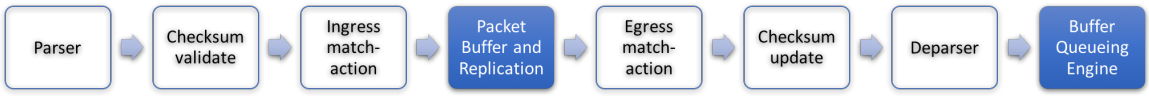

Figure 1. Portable Switch Pipeline

A programmer targeting the PSA is required to instantiate objects for the programmable blocks that conform to these APIs. Note that the programmable block APIs are templatized on user defined headers and metadata. In PSA, the user can define a single metadata type for all controls.

When instantiating the `main` `package` object, the instances corresponding to the programmable blocks are passed as arguments.

2. PSA Data types

2.1. PSA type definitions

These types need to be defined before including the architecture file and the macro protecting them should be defined.

```
typedef bit<unspecified> PortId_t;
typedef bit<unspecified> MulticastGroup_t;
typedef bit<unspecified> PacketLength_t;
typedef bit<unspecified> EgressInstance_t;
typedef bit<unspecified> ParserStatus_t;
typedef bit<unspecified> ParserErrorLocation_t;
typedef bit<unspecified> timestamp_t;

const   PortId_t         PORT_CPU = unspecified;
```

2.2. PSA supported metadata types

```
enum InstanceType_t { NORMAL_INSTANCE, CLONE_INSTANCE }

struct psa_parser_input_metadata_t {
  PortId_t                ingress_port;
  InstanceType_t          instance_type;
}

struct psa_ingress_input_metadata_t {
  PortId_t                ingress_port;
  InstanceType_t          instance_type;  /// Clone or Normal
  /// set by the runtime in the parser, these are not under programmer control
  ParserStatus_t          parser_status;
  ParserErrorLocation_t   parser_error_location;
```

```
  timestamp_t              ingress_timestamp;
}

struct psa_ingress_output_metadata_t {
  PortId_t                 egress_port;
}

struct psa_egress_input_metadata_t {
  PortId_t                 egress_port;
  InstanceType_t           instance_type;  /// Clone or Normal
  EgressInstance_t         instance;       /// instance coming from PRE
  timestamp_t              egress_timestamp;
}
```

2.3. Match kinds

Additional supported match_kind types

```
match_kind {
    range,   /// Used to represent min..max intervals
    selector /// Used for implementing dynamic_action_selection
}
```

2.4. Cloning methods

```
enum CloneMethod_t {
  /// Clone method         Packet source             Insertion point
  Ingress2Ingress,  /// original ingress,            Ingress parser
  Ingress2Egress,    /// post parse original ingress,  Buffering queue
  Egress2Ingress,   /// post deparse in egress,      Ingress parser
  Egress2Egress     /// inout to deparser in egress, Buffering queue
}
```

3. PSA Externs

3.1. Packet Replication Engine

The `PacketReplicationEngine` extern represents the non-programmable part of the PSA pipeline.

Even though the PRE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata. In this specification we opt to define the operations available in the PRE as method invocations. A target backend is responsible for mapping the PRE extern APIs to the appropriate mechanisms for performing these operations in the hardware.

The PRE extern object has no constructor, and thus it cannot be instantiated in the user's P4 program. The architecture instantiates it exactly once, without requiring the user's P4 program to instantiate it. The PRE is made available to the Ingress programmable block using the same mechanism as packet_in. A corresponding Buffering and Queuing Engine (BQE) extern is defined for the Egress pipeline (see 3.2).

Note to implementers: some of these operations may not be implemented as native operations on a certain target. However, the goal is that all operations should be implementable as a combination of native and non-native operations. Target documentation should highlight such implementations, so that application programmers are aware of the potentially significant performance penalties.

Semantics of behavior for multiple calls to PRE APIs

The semantics of calling the PRE APIs is equivalent to setting intrinsic metadata fields/bits and assuming that the PRE looks up the fields in the following order: drop, truncate, multicast, clone, output_port.

Following this semantics, examples of the behaviors are:

- any call to drop in the pipeline will cause the packet, and all potential clone copies (see below) to drop.
- any call to truncate, will cause the packet (and its clones \todo: check) to be truncated.
- multiple calls to send_to_port – the last call in the ingress pipeline sets the output port.
- multiple calls to multicast – the last in the ingress pipeline sets the multicast group
- interleaving send_to_port and multicast – the semantics of multicast is defined as below (https://github.com/p4lang/tutorials/issues/22): if (multicast_group != 0)

  ```
  multicast_to_group(multicast_group);
  ```

  else

  ```
  send_to_port(output_port);
  ```

  From this, it follows that if there is a call that sets the multicast_group, the packet will be multicast to the group that last set the multicast group. Otherwise, the packet will be sent to the port set by send_to_port.
- multiple clone invocations will cause the packet to be cloned to the corresponding port. Any drop call in the pipeline will cause the packet to drop, and no clone will be created (following the analogy with intrinsic metadata bit fields, drop bits are processed before clone bits are looked up).
- resubmit
- recirculate

\TODO: finalize the semantics of calling multiple of the PRE APIs

```
extern PacketReplicationEngine {

  // PacketReplicationEngine(); /// No constructor. PRE is instantiated
                                /// by the architecture.
```

3.1.1. PRE Methods

3.1.1.1. Unicast operation Sends packet to a port.

Targets may implement this operation by setting the appropriate intrinsic metadata or through some other mechanism of configuring the PRE.

The port parameter is the output port. If the port is PORT_CPU the packet will be sent to CPU.

```
void send_to_port (in PortId_t port);
```

3.1.1.2. Multicast operation Sends packet to a multicast group or a port.

Targets may implement this operation by setting the appropriate intrinsic metadata or through some other mechanism of configuring the PRE.

The multicast_group parameter is the multicast group id. The control plane must program the multicast groups through a separate mechanism.

```
void multicast (in MulticastGroup_t multicast_group);
```

3.1.1.3. Drop operation Do not forward the packet.

The PSA implements drop as an operation in the PRE. While the drop operation can be invoked anywhere in the pipeline (ingress or egress), the semantics supported by the PSA is that the drop will be at the end of the pipeline (ingress or egress). Any other operations invoked in the pipeline, whether on the packet or any externs (e.g. counter updates, meter updates, register writes) will execute, until the packet reaches the end of the pipeline. Operations in the egress pipeline will not execute if drop is called in ingress.

```
void drop      ();
```

3.1.2. Clone/recirculation/resubmit

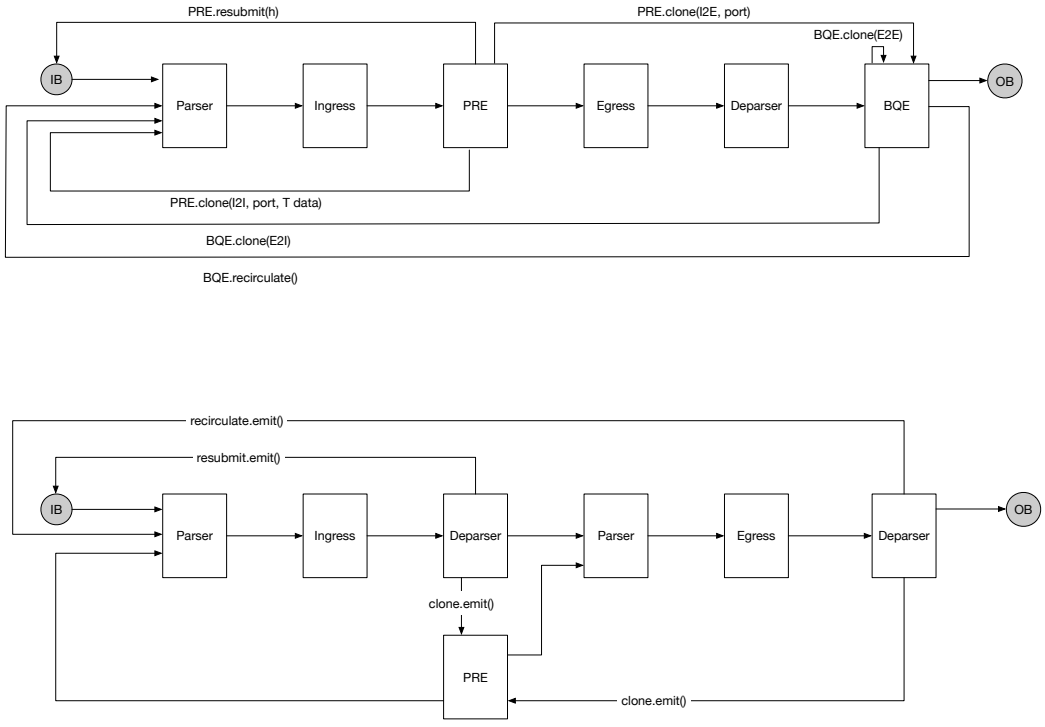

Figure 2. Clone/recirculate/resubmit in PSA

Figure 2 show two proposed architectures for clone/recirculate/resubmit in PSA.

3.1.3. PRE Metadata

```
struct psa_ingress_metadata_for_parser_t {
   bool is_recirc;
}
```

```
struct psa_ingress_metadata_for_pre_t {
   bool clone_ena;
   CloneSpectT clone_spec;
   bool resubmit_ena;
}
struct psa_egress_metadata_for_pre_t {
   bool clone_ena;
   CloneSpecT clone_spec;
   bool recirc_ena;
}
```

3.1.3.1. Clone A PSA implementation provides a clone mechanism to create a copy of the original packet as an independent packet instance. The cloned packet can be a duplicate of the original packet or a copy of the deparsed packet after egress pipeline. The clone mechanism can submit the cloned packet to ingress parser or buffering mechanism. In a PSA implementation, the clone mechanism is implemented in PRE.

The clone mechanism can optionally attach metadata to the cloned packet. A PSA implementation provides the `clone` extern to specify the attached metadata. The `clone` extern provides a `emit` method which accepts the attached metadata of generic type `T`. In a PSA implementation, the metadata is prepended to the cloned packet. It is the responsibility of the programmer to parse the cloned packet correctly to extract the attached metadata. The attached metadata can be of type `header`, `header stack`, `header union` or `struct` of the above types. Invoking the `emit` method multiple times will attach all specified metadata to the same cloned packet. The PSA architecture instantiates the `clone` extern in the ingress and egress deparser. A P4 program can use it in ingress and egress deparser only. It is an error to instantiate the `clone` extern in the P4 control or parser block.

A PSA implementation provides two configuration metadata, `clone_ena` and `clone_spec`, to the PRE to control the cloning mechanism. The `clone_ena` enables/disables the cloning mechanism. If the `clone_ena` bit is set, the clone mechanism generates a cloned packet with the optional attached metadata. If the `clone_ena` bit is unset or unintialized, the clone mechanism is disabled and a cloned packet is not generated even if the `emit` method is invoked in the deparser. The `clone_spec` control the destination of the cloned packet. A common use case is to the send the cloned packet to the control CPU, in which case the `clone_spec` should be set to the value that represents the control CPU port.

The PSA specifies four types of cloning, with the packet sourced from different points in the pipeline and sent back to ingress or to the buffering queue in the egress.

```
extern clone {
  /// Write @hdr into the ingress/egress clone engine.
  /// @T can be a header type, a header stack, a header union, or a struct
  /// containing fields with such types.
  void emit<T>(in T hdr);
}
```

3.1.3.2. Resubmit A PSA implementation provides a resubmit mechanism to resend the original packet to ingress parser for recursive processing. The resubmitted packet is the original packet as seen on the ingress pipeline. The resubmit mechanism sends the original packet to the ingress parser. The resubmit mechanism does not make copies of the original packet. In a PSA implementation, the resubmit mechanism is implemented in PRE.

The resubmit mechanism can optionally attach metadata to the resubmitted packet. A PSA implementation provides the `resubmit` extern to specify the attached metadata. The `resubmit` extern provides a `emit` method which accepts the attached metadata of generic type `T`. In a PSA

implementation, the metadata is prepended to the resubmitted packet. It is the responsibility of the programmer to parse the resubmitted packet correctly to extract the attached metadata. The attached metadata can be of type `header`, `header stack`, `header union` or `struct` of the above types. Invoking the `emit` method multiple times will attach all specified metadata to the same resubmitted packet. The PSA architecture instantiates the `resubmit` extern in the ingress deparser. A P4 program can use it in ingress deparser only. It is an error to instantiate the `resubmit` extern in the P4 control or parser block.

A PSA implementation provides a configuration bit `resubmit_ena` to the PRE to enable the resubmit mechanism. If the `resubmit_ena` bit is set, the resubmit mechanism resends the original packet with the optional attached metadata. If the `resubmit_ena` bit is unset or unintialized, the resubmit mechanism is disabled and the original packet is not resubmitted even if the `emit` method is invoked in the deparser.

```
extern resubmit {
  /// Write @hdr into the ingress packet buffer.
  /// @T can be a header type, a header stack, a header union or a struct
  /// containing fields with such types.
  void emit<T>(in T hdr);
}
```

3.1.3.3. Recirculate A PSA implementation provides a recirculation mechanism to send a deparsed packet from egress pipeline to ingress parser for recursive processing. The recirculated packet is the deparsed packet after the egress pipeline. The recirculation mechanism does not make copies of the original packet. It sends the deparsed packet from egress back to the ingress parser. In a PSA implementation, the recirculation mechanism is implemented in PRE.

The recirculation mechanism can optionally attach metadata to the recirculated packet. A PSA implementation provides the `recirculate` extern to specify the attached metadata. The `emit` method in the `recirculate` extern accepts an attached metadata of generic type `T`. In a PSA implementation, the metadata is prepended to the recirculated packet. It is the responsibility of the programmer to parse the recirculated packet correctly to extract the attached metadata. The attached metadata can be of type `header`, `header stack`, `header union` or `struct` of the above types. Invoking the `emit` method multiple times will attach all specified metadata to the same recirculated packet. The PSA architecture instantiates the `recirculate` extern in the ingress deparser. A P4 program can use it in egress deparser only. It is an error to instantiate the `recirculate` extern in a P4 control or parser block.

A PSA implementation provides a configuration bit `recirc_ena` to the PRE to enable the recirculation mechanism. If the `recirc_ena` bit is set, the recirculation mechanism sends back the deparsed packet with the optional attached metadata to the ingress parser. If the `recirc_ena` bit is unset or unintialized, the recirculation mechanism is disabled and the deparsed packet is not recirculated even if the `emit` method is invoked in the deparser.

One possible implementaion of the `recirc_ena` bit is to set the `egress_port` metadata to a dedicated recirculation port number. A PSA implementation sends the deparsed packet to the dedicated recirculation port to recirculate the packet.

A PSA implementation provides a metadata bit `is_recirc` as the input metadata to the parser to indicate if a packet is a recirculation packet. The `is_recirc` bit is true if the packet is a recirculation packet.

One possible implementation of the `is_recirc` bit is to set the bit based on the `ingress_port` metadata of the received packet. If the `ingress_port` metadata is equal to the dedicated recirculation port number, then the `is_recirc` bit is set.

```
extern recirculate {
  /// Write @hdr into the egress packet.
  /// @T can be a header type, a header stack, a header union or a struct
```

```
  /// containing fields with such types.
  void emit(in T hdr);
}
```

3.1.3.4. Truncate operation Truncate the payload of the outgoing packet to the specified length.

The length parameter represents the desired payload length in bytes. To remove the payload one may invoke `truncate(0)`.

```
void truncate(in bit<32> length);
```

3.2. Buffering Queuing Engine

The BufferingQueueingEngine extern represents the the other non-programmable part of the PSA pipeline (after Egress).

Even though the BQE can not be programmed using P4, it can be configured both directly using control plane APIs and by setting intrinsic metadata. In this specification we opt to define the operations available in the BQE as method invocations. A target backend is responsible for mapping the BQE extern APIs to the appropriate mechanisms for performing these operations in the hardware.

The BQE extern object has no constructor, and thus it cannot be instantiated in the user's P4 program. The architecture instantiates it exactly once, without requiring the user's P4 program to instantiate it. The BQE is made available to the Egress programmable block using the same mechanism as packet_in. A corresponding Packet Replication Engine (PRE) extern is defined for the Ingress pipeline (see 3.1).

Note to implementers: some of these operations may not be implemented as primitive operations on a certain target. However, the goal is that all operations should be implementable as a combination of other operations. Target documentation should highlight such implementations, so that application programmers are aware of the potentially significant performance penalties.

The ordering semantics of multiple calls to BQE APIs is identical to the semantics ordering of PRE invocations, for the subset of functions supported in the BQE.

```
extern BufferingQueueingEngine {

  // BufferingQueueingEngine(); /// No constructor. BQE is instantiated
                                /// by the architecture.
```

3.2.1. BQE Methods

3.2.1.1. Drop operation Do not forward the packet.

The PSA implements drop as an operation in the BQE. While the drop operation can be invoked anywhere in the ingress pipeline, the semantics supported by the PSA is that the drop will be at the end of the pipeline (ingress or egress).

```
void drop      ();
```

3.2.1.2. Truncate operation Truncate the outgoing packet to the specified length

The length parameter represents the packet length.

```
void truncate(in bit<32> length);
```

3.3. Hashes

Supported hash algorithms:

```
enum HashAlgorithm {
  identity,
  crc32,
  crc32_custom,
  crc16,
  crc16_custom,
  ones_complement16,  /// One's complement 16-bit sum used for IPv4 headers,
                      /// TCP, and UDP.
  target_default      /// target implementation defined
}
```

3.3.1. Hash function

Example usage:

```
parser P() {
  Hash<bit<16>>(HashAlgorithm.crc16) h;
  bit<16> hash_value = h.getHash(buffer);
}
```

Parameters:

- algo The algorithm to use for computation (see 3.3).
- O The type of the return value of the hash.

```
extern Hash<O> {
  /// Constructor
  Hash(HashAlgorithm algo);

  /// Compute the hash for data.
  /// @param data The data over which to calculate the hash.
  /// @return The hash value.
  O getHash<D>(in D data);

  /// Compute the hash for data, with modulo by max, then add base.
  /// @param base Minimum return value.
  /// @param data The data over which to calculate the hash.
  /// @param max The hash value is divided by max to get modulo.
  ///        An implementation may limit the largest value supported,
  ///        e.g. to a value like 32, or 256.
  /// @return (base + (h % max)) where h is the hash value.
  O getHash<T, D>(in T base, in D data, in T max);
}
```

TBD: Should there be a `const` defined that specifies the maximum allowed value of `max` parameter?

3.4. Checksum computation

Checksums and hash value generators are examples of functions that operate on a stream of bytes from a packet to produce an integer. The integer may be used, for example, as an integrity check for a packet or as a means to generate a pseudo-random value in a given range on a packet-by-packet or flow-by-flow basis.

Parameters:

- W The width of the checksum

```
extern Checksum<W> {
  Checksum(HashAlgorithm hash);          /// constructor
  void clear();              /// prepare unit for computation
  void update<T>(in T data); /// add data to checksum
  void remove<T>(in T data); /// remove data from existing checksum
  W    get();               /// get the checksum for data added since last clear
}
```

3.5. Counters

Counters are a mechanism for keeping statistics. The control plane can read counter values. A P4 program cannot read counter values, only update them. If you wish to implement a feature involving sequence numbers in packets, for example, use Registers instead (Section 3.7).

Direct counters are counters associated with a particular P4 table, and are implemented by the extern `DirectCounter`. There are also indexed counters, which are implemented by the extern `Counter`. The primary differences between direct counters and indexed counters are:

- Number of independently updatable counter values:
  - A single instantiation of a direct counter always contains as many independent counter values as the number of entries in the table with which it is associated (TBD: see below for what this means for tables that use action profiles).
  - You must specify the number of independent counter values for an indexed counter when instantiating it. This number of counters need not be the same as the size of any table.
- Where counter updates are allowed in the P4 program:
  - For a direct counter, you may only invoke its `count` method from inside the actions of the table with which it is associated, and this always updates the counter value associated with the matching table entry.
  - For an indexed counter, you may invoke its `count` method anywhere in the P4 program where extern object method invocations are permitted (e.g. inside actions, or directly inside a control's `apply` block), and every such invocation must specify the index of the counter value to be updated.

Counters are only intended to support packet counters and byte counters, or a combination of both called `packets_and_bytes`. The byte counts are always increased by some measure of the packet length, where the packet length used might vary from one PSA implementation to another. For example, one implementation might use the Ethernet frame length, including the Ethernet header and FCS bytes, as the packet arrived on a physical port. Another might not include the FCS bytes in its definition of the packet length. Another might only include the Ethernet payload length. Each PSA implementation should document how it determines the packet length used for byte counter updates.

If you wish to keep counts of other quantities, or to have more precise control over the packet length used in a byte counter, you may use Registers to achieve that (Section 3.7).

3.5.1. Counter types

```
enum CounterType_t {
    packets,
    bytes,
    packets_and_bytes
}
```

3.5.2. Counter

```
/// Indirect counter with n_counters independent counter values, where
/// every counter value has a data plane size specified by type W.

extern Counter<W, S> {
  Counter(bit<32> n_counters, CounterType_t type);
  void count(in S index);

  /*
  /// The control plane API uses 64-bit wide counter values.  It is
  /// not intended to represent the size of counters as they are
  /// stored in the data plane.  It is expected that control plane
  /// software will periodically read the data plane counter values,
  /// and accumulate them into larger counters that are large enough
  /// to avoid reaching their maximum values for a suitably long
  /// operational time.  A 64-bit byte counter increased at maximum
  /// line rate for a 100 gigabit port would take over 46 years to
  /// wrap.

  @ControlPlaneAPI
  {
    bit<64> read      (in S index);
    bit<64> sync_read (in S index);
    void set          (in S index, in bit<64> seed);
    void reset        (in S index);
    void start        (in S index);
    void stop         (in S index);
  }
  */
}
```

See below for pseudocode of an example implementation for the Counter extern.

The example implementation for `next_counter_value` is not intended to restrict PSA implementations. In particular, the storage format for `packets_and_bytes` type counters is just one example of how it could be done. Implementations are free to store state in other ways, as long as the control plane API returns the correct packet and byte count values.

Two common techniques for counter implementations in the data plane are:

- wrap around counters
- saturating counters, that 'stick' at their maximum possible value, without wrapping around.

This specification does not mandate any particular approach in the data plane. Implementations should strive to avoid losing information in counters. One common implementation technique is to implement an atomic "read and clear" operation in the data plane that can be invoked by the control plane software. The control plane software invokes this operation frequently enough to prevent counters from ever wrapping or saturating, and adds the values read to larger counters in driver memory.

```
Counter(bit<32> n_counters, CounterType_t type) {
    this.num_counters = n_counters;
    this.counter_vals = new array of size n_counters, each element with type W;
    this.type = type;
    if (this.type == CounterType_t.packets_and_bytes) {
        // Packet and byte counts share storage in the same counter
```

```
        // state.  Should we have a separate constructor with an
        // additional argument indicating how many of the bits to use
        // for the byte counter?
        W shift_amount = TBD;
        this.shifted_packet_count = ((W) 1) << shift_amount;
        this.packet_count_mask = (~((W) 0)) << shift_amount;
        this.byte_count_mask = ~this.packet_count_mask;
    }
}

W next_counter_value(W cur_value, CounterType_t type) {
    if (type == CounterType_t.packets) {
        return (cur_value + 1);
    }
    // Exactly which packet bytes are included in packet_len is
    // implementation-specific.
    PacketLength_t packet_len = <packet length in bytes>;
    if (type == CounterType_t.bytes) {
        return (cur_value + packet_len);
    }
    // type must be CounterType_t.packets_and_bytes
    // In type W, the least significant bits contain the byte
    // count, and most significant bits contain the packet count.
    // This is merely one example storage format.  Implementations
    // are free to store packets_and_byte state in other ways, as
    // long as the control plane API returns the correct separate
    // packet and byte count values.
    W next_packet_count = ((cur_value + this.shifted_packet_count) &
                           this.packet_count_mask);
    W next_byte_count = (cur_value + packet_len) & this.byte_count_mask;
    return (next_packet_count | next_byte_count);
}

void count(in S index) {
    if (index < this.num_counters) {
        this.counter_vals[index] = next_counter_value(this.counter_vals[index],
                                                      this.type);
    } else {
        // No counter_vals updated if index is out of range.
        // See below for optional debug information to record.
    }
}
```

Optional debugging information that may be kept if an `index` value is out of range includes:

- Number of times this occurs.
- A FIFO of the first N out-of-range index valus that occur, where N is implementation-defined (e.g. it might only be 1). Extra information to identify which `count()` method call in the P4 program had the out-of-range `index` value is also recommended.

3.5.3. Direct Counter

```
extern DirectCounter<W> {
  DirectCounter(CounterType_t type);
  void count();

  /*
  @ControlPlaneAPI
  {
    W    read<W>      (in TableEntry key);
    W    sync_read<W> (in TableEntry key);
    void set          (in W seed);
    void reset        (in TableEntry key);
    void start        (in TableEntry key);
    void stop         (in TableEntry key);
  }
  */
}
```

A `DirectCounter` instance must appear in the list of values of the `psa_direct_counters` table attribute for exactly one table. We call this table the `DirectCounter` instance's "owner". It is an error to call the `count` method for a `DirectCounter` instance anywhere except inside an action of its owner table.

The counter value updated by an invocation of `count` is always the one associated with the table entry that matched.

TBD: How to describe which counter value is updated for tables with action profiles and direct counters? Or should this combination even be allowed?

An action of an owner table need not have `count` method calls for all of the `DirectCounter` instances that the table owns. You must use an explicit `count()` method call on a `DirectCounter` to update it, otherwise its state will not change.

An example implementation for the `DirectCounter` extern is essentially the same as the one for `Counter`. Since there is no `index` parameter to the `count` method, there is no need to check for whether it is in range.

The rules here mean that an action that calls `count` on a `DirectCounter` instance may only be an action of that instance's one owner table. If you want to have a single action `A` that can be invoked by multiple tables, you can still do so by having a unique action for each such table with a `DirectCounter`, where each such action in turn calls action `A`, in addition to any `count` invocations they have.

A `DirectCounter` instance must have a counter value associated with its owner table that is updated when there is a default action assigned to the table, and a search of the table results in a miss. If there is no default action assigned to the table, then there need not be any counter updated when a search of the table results in a miss.

By "a default action is assigned to a table", we mean that either the table has a `default_action` table property with an action assigned to it in the P4 program, or the control plane has made an explicit call to assign the table a default action. If neither of these is true, then there is no default action assigned to the table.

TBD: Verify that the method of reading this default action counter state is documented for the control plane API. I believe that Antonin Bas said that it can be accessed using the same API call used to read a `DirectCounter` value associated with a table entry, except that the key in the API call should be empty.

TBD: Should a single table be restricted to have at most one DirectCounter associated with it, or should it be allowed to have more than one?

3.5.4. Example program using counters

The following partial P4 program demonstrates the instantiation and updating of `Counter` and `DirectCounter` externs.

```
typedef bit<48> ByteCounter_t;
typedef bit<32> PacketCounter_t;
typedef bit<80> PacketByteCounter_t;

const PortId_t NUM_PORTS = 512;

struct headers {
    ethernet_t       ethernet;
    ipv4_t           ipv4;
}

control ingress(inout headers hdr,
                inout metadata user_meta,
                PacketReplicationEngine pre,
                in  psa_ingress_input_metadata_t  istd,
                out psa_ingress_output_metadata_t ostd)
{
    Counter<ByteCounter_t, PortId_t>((bit<32>) NUM_PORTS, CounterType_t.bytes)
        port_bytes_in;
    DirectCounter<PacketByteCounter_t>(CounterType_t.packets_and_bytes)
        per_prefix_pkt_byte_count;

    action next_hop(PortId_t oport) {
        per_prefix_pkt_byte_count.count();
        ostd.egress_port = oport;
    }
    action default_route_drop() {
        per_prefix_pkt_byte_count.count();
        pre.drop();
    }
    table ipv4_da_lpm {
        key = { hdr.ipv4.dstAddr: lpm; }
        actions = {
            next_hop;
            default_route_drop;
        }
        default_action = default_route_drop;
        //psa_direct_counters = {
        //    // table ipv4_da_lpm owns this DirectCounter instance
        //    per_prefix_pkt_byte_count;
        //}
    }
    apply {
        port_bytes_in.count(istd.ingress_port);
        ostd.egress_port = 0;
        if (hdr.ipv4.isValid()) {
            ipv4_da_lpm.apply();
        }
```

```
    }
}

control egress(inout headers hdr,
               inout metadata user_meta,
               BufferingQueueingEngine bqe,
               in  psa_egress_input_metadata_t  istd)
{
    Counter<ByteCounter_t, PortId_t>((bit<32>) NUM_PORTS, CounterType_t.bytes)
        port_bytes_out;
    apply {
        // By doing these stats updates on egress, as long as IP
        // multicast replication happens in the packet buffer, this
        // update will occur once for each copy made, which in this
        // example is intentional.
        port_bytes_out.count(istd.egress_port);
    }
}
```

3.6. Meters

Meters (RFC 2698) are a more complex mechanism for keeping statistics about packets, most often used for dropping or "marking" packets that exceed an average packet or bit rate. To mark a packet means to change one or more of its quality of service values in packet headers such as the 802.1Q PCP (priority code point) or DSCP (differentiated service code point) bits within the IPv4 or IPv6 type of service byte. The meters specified in the PSA are 3-color meters.

PSA meters do not require any particular drop or marking actions, nor do they automatically implement those behaviors for you. Meters keep enough state, and update their state during `execute()` method calls, in such a way that they return a `GREEN` (also known as conform), `YELLOW` (exceed), or `RED` (violate) result. See RFC 2698 for details on the conditions under which one of these three results is returned. The P4 program is responsible for examining that returned result, and making changes to packet forwarding behavior as a result.

RFC 2698 describes "color aware" and "color blind" variations of meters. The `Meter` and `DirectMeter` externs implement both. The only difference is in which `execute` method you use when updating them. See the comments on the `extern` definitions below.

Similar to counters, there are two flavors of meters: indexed and direct. (Indexed) meters are addressed by index, while direct meters always update a meter state corresponding to the matched table entry or action, and from the control plane API are addressed using P4Runtime table entry as key.

There are many other similarities between counters and meters, including:

- The number of independently updatable meter values.
- Where meter updates are allowed in a P4 program.
- For `bytes` type meters, the packet length used in the update is determined by the PSA implementation, and can vary from one PSA implementation to another.

Further similarities between direct counters and direct meters include:

- `DirectMeter` `execute` method calls must be performed within actions invoked by the table that owns the `DirectMeter` instance. It is optional for such an action to call the `execute` method.
- There must be a meter state associated with a `DirectMeter` instance's owner table, that can be updated when the table result is a miss. As for a `DirectCounter`, this state only needs to exist if a default action is assigned to the table.

The table attribute to specify that a table owns a `DirectMeter` instance is `psa_direct_meters`. The value of this table attribute is a list of meter instances.

As for counters, if you call the `execute(idx)` method on an indexed meter and `idx` is at least the number of meter states, so `idx` is out of range, no meter state is updated. The `execute` call still returns a value of type `MeterColor_t`, but the value is undefined – programs that wish to have predictable behavior across implementations must not use the undefined value in a way that affects the output packet or other side effects. The example code below shows one way to achieve predictable behavior. Note that this undefined behavior cannot occur if the value of `n_meters` of an indexed meter is 2^W, and the type `S` used to construct the meter is `bit<W>`, since the index value could never be out of range.

```
#define METER1_SIZE 100
Meter<bit<7>>(METER1_SIZE, MeterType_t.bytes) meter1;
bit<7> idx;
MeterColor_t color1;

// ... later ...

if (idx < METER1_SIZE) {
    color1 = meter1.execute(idx, MeterColor_t.GREEN);
} else {
    // If idx is out of range, use a default value for color1.  One
    // may also choose to store an error flag in some metadata field.
    color1 = MeterColor_t.RED;
}
```

Any implementation will have a finite range that can be specified for the Peak Burst Size and Committed Burst Size. An implementation should document the maximum burst sizes they support, and if the implementation internally truncates the values that the control plane requests to something more coarse than any number of bytes, that should also be documented. It is recommended that the maximum burst sizes be allowed as large as the number of bytes that can be transmitted across the implementation's maximum speed port in 100 milliseconds.

Implementations will also have finite ranges and precisions that they support for the Peak Information Rate and Committed Information Rate. An implementation should document the maximum rate it supports, as well as the precision it supports for implementing requested rates. It is recommended that the maximum rate supported be at least the rate of the implementation's fastest port, and that the actual implemented rate should always be within plus or minus 0.1% of the requested rate.

3.6.1. Meter types

```
enum MeterType_t {
    packets,
    bytes
}
```

3.6.2. Meter colors

```
enum MeterColor_t { RED, GREEN, YELLOW };
```

3.6.3. Meter

```
// Indexed meter with n_meters independent meter states.

extern Meter<S> {
  Meter(bit<32> n_meters, MeterType_t type);

  // Use this method call to perform a color aware meter update (see
  // RFC 2698). The color of the packet before the method call was
  // made is specified by the color parameter.
  MeterColor_t execute(in S index, in MeterColor_t color);

  // Use this method call to perform a color blind meter update (see
  // RFC 2698).  It may be implemented via a call to execute(index,
  // MeterColor_t.GREEN), which has the same behavior.
  MeterColor_t execute(in S index);

  /*
  @ControlPlaneAPI
  {
    reset(in MeterColor_t color);
    setParams(in S index, in MeterConfig config);
    getParams(in S index, out MeterConfig config);
  }
  */
}
```

3.6.4. Direct Meter

```
extern DirectMeter {
  DirectMeter(MeterType_t type);
  // See the corresponding methods for extern Meter.
  MeterColor_t execute(in MeterColor_t color);
  MeterColor_t execute();

  /*
  @ControlPlaneAPI
  {
    reset(in TableEntry entry, in MeterColor_t color);
    void setConfig(in TableEntry entry, in MeterConfig config);
    void getConfig(in TableEntry entry, out MeterConfig config);
  }
  */
}
```

3.7. Registers

Registers are stateful memories whose values can be read and written during packet forwarding under the control of the P4 program. They are similar to counters and meters in that their state can be modified as a result of processing packets, but they are far more general in the behavior they can implement.

Although you may not use register contents directly in table match keys, you may use the `read()` method call on the right-hand side of an assignment statement, which retrieves the current value of

the register. You may copy the register value into metadata, and it is then available for matching in subsequent tables.

A simple usage example might be to verify that a "first packet" was seen for a particular type of flow. A register cell would be allocated to the flow, initialized to "clear". When the protocol signaled a "first packet", the table would match on this value and update the flow's cell to "marked". Subsequent packets in the flow could would be mapped to the same cell; the current cell value would be stored in metadata for the packet and a subsequent table could check that the flow was marked as active.

```
extern Register<T, S> {
  Register(bit<32> size);
  T    read  (in S index);
  void write (in S index, in T value);

  /*
  @ControlPlaneAPI
  {
    T    read<T>      (in S index);
    void set          (in S index, in T seed);
    void reset        (in S index);
  }
  */
}
```

Another example using registers is given below. It implements a packet and byte counter, where the byte counter can be updated by a packet length specified in the P4 program, rather than one chosen by the PSA implementation.

```
const PortId_t NUM_PORTS = 512;

// It would be more convenient to use a struct type to represent the
// state of a combined packet and byte count, and many other compound
// values one might wish to store in a Register instance.  However,
// the latest p4test as of 2017-Aug-13 does not allow a struct type to
// be returned from a method call like Register.read().

#define PACKET_COUNT_WIDTH 32
#define BYTE_COUNT_WIDTH 48
//#define PACKET_BYTE_COUNT_WIDTH (PACKET_COUNT_WIDTH + BYTE_COUNT_WIDTH)
#define PACKET_BYTE_COUNT_WIDTH 80

#define PACKET_COUNT_RANGE (PACKET_BYTE_COUNT_WIDTH-1):BYTE_COUNT_WIDTH
#define BYTE_COUNT_RANGE (BYTE_COUNT_WIDTH-1):0

typedef bit<PACKET_BYTE_COUNT_WIDTH> PacketByteCountState_t;

action update_pkt_ip_byte_count (inout PacketByteCountState_t s,
                                 in bit<16> ip_length_bytes)
{
    s[PACKET_COUNT_RANGE] = s[PACKET_COUNT_RANGE] + 1;
    s[BYTE_COUNT_RANGE] = (s[BYTE_COUNT_RANGE] +
                           (bit<BYTE_COUNT_WIDTH>) ip_length_bytes);
}
```

```
control ingress(inout headers hdr,
                inout metadata user_meta,
                PacketReplicationEngine pre,
                in  psa_ingress_input_metadata_t  istd,
                out psa_ingress_output_metadata_t ostd)
{
    Register<PacketByteCountState_t, PortId_t>((bit<32>) NUM_PORTS)
        port_pkt_ip_bytes_in;

    apply {
        ostd.egress_port = 0;
        if (hdr.ipv4.isValid()) {
            @atomic {
                PacketByteCountState_t tmp;
                tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);
                update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);
                port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);
            }
        }
    }
}
```

Note the use of the `@atomic` annotation in the block enclosing the `read()` and `write()` method calls on the `Register` instance. It is expected to be common that register accesses will need the `@atomic` annotation around portions of your program in order to behave as you desire. As stated in the P4_16 specification, without the `@atomic` annotation in this example, an implementation is allowed to process two packets `P1` and `P2` in parallel, and perform the register access operations in this order:

```
// Possible order of operations for the example program if the
// @atomic annotation is _not_ used.

tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);  // for packet P1
tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);  // for packet P2

// At this time, if P1 and P2 came from the same ingress_port,
// each of their values of tmp are identical.

update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);    // for packet P1
update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);    // for packet P2

port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);  // for packet P1
port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);  // for packet P2
// The write() from packet P1 is lost.
```

Since different implementations may have different upper limits on the complexity of code that they will accept within an `@atomic` block, we recommend you keep them as small as possible, subject to maintaining your desired correct behavior.

Individual counter and meter method calls need not be enclosed in `@atomic` blocks to be safe – they guarantee atomic behavior of their individual method calls, without losing any updates.

As for indexed counters and meters, access to an index of a register that is at least the size of the register is out of bounds. An out of bounds write has no effect on the state of the system. An out of bounds read returns an undefined value. See the example in Section 3.6 for one way to write code to guarantee avoiding this undefined behavior. Out of bounds register accesses are impossible

for a register instance with type S declared as bit<W> and size 2^W entries.

3.8. Random

The random extern provides a reliable, target specific number generator in the min .. max range.

The set of distributions supported by the Random extern. \TODO: should this be removed in favor of letting the extern return whatever distribution is supported by the target?

```
enum RandomDistribution {
  PRNG,
  Binomial,
  Poisson
}

extern Random<T> {
  Random(RandomDistribution dist, T min, T max);
  T read();

  /*
  @ControlPlaneAPI
  {
    void reset();
    void setSeed(in T seed);
  }
  */
}
```

3.9. Action Profile

Action profiles are used as table implementation attributes.

Action profiles implement a mechanism to populate table entries with actions and action data. The only data plane operation required is to instantiate this extern. When the control plane adds entries (members) into the extern, they are essentially populating the corresponding table entries.

```
extern ActionProfile {
  /// Construct an action profile of 'size' entries
  ActionProfile(bit<32> size);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member    (action_ref, action_data);
     void         delete_member (entry_handle);
     entry_handle modify_member (entry_handle, action_ref, action_data);
  }
  */
}
```

3.10. Action Selector

Action selectors are used as table implementation attributes.

Action selectors implement another mechanism to populate table entries with actions and action data. They are similar to action profiles, with additional support to define groups of entries. Action selectors require a hash algorithm to select members in a group. The only data plane operation

required is to instantiate this extern. When the control plane adds entries (members) into the extern, they are essentially populating the corresponding table entries.

```
extern ActionSelector {
  /// Construct an action selector of 'size' entries
  /// @param algo hash algorithm to select a member in a group
  /// @param size number of entries in the action selector
  /// @param outputWidth size of the key
  ActionSelector(HashAlgorithm algo, bit<32> size, bit<32> outputWidth);

  /*
  @ControlPlaneAPI
  {
     entry_handle add_member        (action_ref, action_data);
     void         delete_member     (entry_handle);
     entry_handle modify_member     (entry_handle, action_ref, action_data);
     group_handle create_group      ();
     void         delete_group      (group_handle);
     void         add_to_group      (group_handle, entry_handle);
     void         delete_from_group (group_handle, entry_handle);
  }
  */
}
```

3.11. Packet Generation

\TODO: is generating a new packet and sending it to the stream or is it adding a header to the current packet and sending it to the stream (copying or redirecting).

```
extern Digest<T> {
  Digest(PortId_t receiver); /// define a digest stream to receiver
  void emit(in T data);      /// emit data into the stream

  /*
  @ControlPlaneAPI
  {
  // TBD
  // If the type T is a named struct, the name should be used
  // to generate the control-plane API.
  }
  */
}
```

3.12. Parser Value Sets

A parser value set is a named set of values that may be used during packet header parsing time to make decisions. You may use control plane API calls to add values to a set, and remove values from a set, at run time, much like P4 tables. Unlike tables, they may not have actions associated with them. They may only be used to determine whether a particular value is in the set, returning a Boolean value. That Boolean value can then be used in a `select` statement to control parsing (see examples below).

```
extern ValueSet<D> {
    ValueSet(int<32> size);
```

```
    bool is_member(in D data);

    /*
    @ControlPlaneAPI
    message ValueSetEntry {
        uint32 value_set_id = 1;
        // FieldMatch allows specification of exact, lpm, ternary, and
        // range matching on fields for tables, and these options are
        // permitted for the ValueSet extern as well.
        repeated FieldMatch match = 2;
    }

    // ValueSetEntry should be added to the 'message Entity'
    // definition, inside its 'oneof Entity' list of possibilities.
    */
}
```

The control plane API excerpt above is intended to be added as part of the P4Runtime API[1].

The control plane API for a `ValueSet` is similar to that of a table, except only match fields may be specified, with no actions. This includes API calls that specify ternary or range matching, although for `ValueSet`s these do not require specifying any priority values, since the only result of a `ValueSet` `is_member` call is "in the set" or "not in the set".

If a PSA target can do so, it should implement control plane API calls involving ternary or range matching using ternary or range matching capabilities in the target, consuming the minimal table entries possible.

However, a PSA target is allowed to implement such control plane API calls by "expanding" them into as many exact match entries as needed to have the same behavior. For example, a control plane API call adding all values in the range 5 through 8 may be implemented as adding the four separate exact match values 5, 6, 7, and 8.

The parser definition below shows an example that uses two `ValueSet` instances called `tpid_types` and `trill_types`.

```
parser ParserImpl(packet_in buffer,
                  out headers parsed_hdr,
                  inout metadata user_meta,
                  in psa_parser_input_metadata_t istd)
{
    ValueSet<bit<16>>(4) tpid_types;
    ValueSet<bit<16>>(2) trill_types;
    state start {
        buffer.extract(parsed_hdr.ethernet);
        transition select(parsed_hdr.ethernet.etherType) {
            0x0800: parse_ipv4;
            0x86DD: parse_ipv6;
            default: dispatch_tpid_value_set;
        }
    }
    state dispatch_tpid_value_set {
        bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_tpid) {
            true: parse_vlan_tag;
```

[1] The P4Runtime API, defined as a Google Protocol Buffer `.proto` file, can be found at https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

```
            default: dispatch_trill_value_set;
        }
    }
    state dispatch_trill_value_set {
        bool is_trill = trill_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_trill) {
            true: parse_trill;
            default: accept;
        }
    }
    state parse_vlan_tag {
        // extract VLAN 802.1Q header here
        transition accept;
    }
    state parse_trill {
        // extract TRILL header here
        transition accept;
    }
    state parse_ipv4 {
        transition accept;
    }
    state parse_ipv6 {
        transition accept;
    }
}
```

The second example (below) has the same parsing behavior as the example above, but combines the two parse states `dispatch_tpid_value_set` and `dispatch_trill_value_set` into one.

```
    state dispatch_tpid_value_set {
        bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);
        bool is_trill = trill_types.is_member(parsed_hdr.ethernet.etherType);
        transition select(is_tpid, is_trill) {
            (true,     _): parse_vlan_tag;
            (false, true): parse_trill;
            default: accept;
        }
    }
```

The third example (below) demonstrates one way to have a `ValueSet` that matches on multiple fields, by making the type `D` a `struct` containing multiple bit vectors.

```
struct CustomValueSet1_t {
    bit<16> etherType;
    bit<8>  partialMacAddress;
}

parser ParserImpl(packet_in buffer,
                  out headers parsed_hdr,
                  inout metadata user_meta,
                  in psa_parser_input_metadata_t istd)
{
    ValueSet<CustomValueSet1_t>(2) trill_types;

    state dispatch_tpid_value_set {
```

```
        bool is_trill =
            trill_types.is_member({parsed_hdr.ethernet.etherType,
                                   parsed_hdr.ethernet.dstAddr[7:0]});
        transition select(is_trill) {
            true: parse_vlan_tag;
            default: accept;
        }
    }

    // ... etc.
```

A PSA compliant implementation is not required to support any use of a `ValueSet` `is_member` method call return value, other than directly inside of a `select` expression. For example, a program fragment like the one shown below may be rejected, and thus P4 programmers striving for maximum portability should avoid writing such code.

```
bool is_tpid = tpid_types.is_member(parsed_hdr.ethernet.etherType);

is_tpid = is_tpid && (parsed_hdr.ethernet.dstAddr[47:40] == 0xfe);
transition select(is_tpid) {
    // ...
```

3.13. Timestamps

A PSA implementation provides an `ingress_timestamp` value for every packet in the ingress control block, as a field in the struct with type `psa_ingress_input_metadata_t`. This timestamp should be close to the time that the first bit of the packet arrived to the device, or alternately, to the time that the device began parsing the packet. This timestamp is *not* automatically included with the packet in the egress control block. A P4 program wishing to use the value of `ingress_timestamp` in egress code must copy it to a user-defined metadata field that reaches egress.

A PSA implementation also provides an `egress_timestamp` value for every packet in the egress control block, as a field of the struct with type `psa_egress_input_metadata_t`.

One expected use case for timestamps is to store them in tables or `Register` instances to implement checking for timeout events for protocols, where precision on the order of milliseconds is sufficient for most protocols.

Another expected use case is INT (Inband Network Telemetry[2]), where precision on the order of microseconds or smaller is necessary to measure queueing latencies that differ by those amounts. It takes only 0.74 microseconds to transmit a 9 Kbyte Ethernet jumbo frame on a 100 gigabit per second link.

For these applications, it is recommended that an implementation's timestamp increments at least once every microsecond. Incrementing once per clock cycle in an ASIC or FPGA implementation would be a reasonable choice. The timestamp should increment at a constant rate over time. For example, it should not be a simple count of clock cycles in a device that implements dynamic frequency scaling[3].

Timestamps are of type `timestamp_t`, which is type `bit<W>` for a value of `W` defined by the implementation. Timestamps are expected to wrap around during the normal passage of time. It is recommended that an implementation pick a rate of advance and a bit width such that wrapping around occurs at most once every hour. Making the wrap time this long (or longer) makes timestamps more useful for several use cases.

- Checking for timeouts of protocol hello / keep-alive traffic that is on the order of seconds or minutes.

[2] http://p4.org/p4/inband-network-telemetry
[3] https://en.wikipedia.org/wiki/Dynamic_frequency_scaling

- If timestamps are placed into packets without converting them to other formats, then external data analysis systems using those timestamps will in many cases need to do so, e.g. to compare timestamps stored in packets by different PSA devices. These systems will need different formulas and/or parameters to perform this conversion for each wrap period, or to add extra external time references to the recorded data. The extra data required for accurate conversion is lower, and the likelihood of conversion mistakes is lower, if the timestamp values wrap less often.
- If timestamps are converted to other formats within a P4 program, it will need access to parameters that are likely to change every wrap time, e.g. at least a "base value" to add some calculated value to. A straightforward way to do this requires the control plane to update these values at least once or twice per timestamp wrap time.
- Programs that wish to use `(egress_timestamp - ingress_timestamp)` to calculate the queueing latency experienced by a packet need the wrap time to exceed the maximum queueing latency.

Examples of the number of bits required for wrap times of at least one hour:

- A 32-bit timestamp advancing by 1 per microsecond takes 1.19 hours to wrap.
- A 42-bit timestamp advancing by 1 per nanosecond takes 1.22 hours to wrap.

A PSA implementation is not required to implement time synchronization, e.g. via PTP[4] or NTP[5].

TBD: This text has been written assuming that it is more important for timestamps to be increasing at a constant rate, with no sudden "jumps" due to time synchronization events. Is this what people want from timestamps?

TBD: Some time synchronization methods avoid sudden "jumps" by temporarily speeding up or slowing down the rate of increase by a small percentage, until the desired synchronization is achieved. (TBD: which ones? citation?). Would anyone mind if PSA implementations were allowed to do this with their timestamp values?

The control plane API excerpt below is intended to be added as part of the P4Runtime API[1].

```
// The TimestampInfo and Timestamp messages should be added to the
// "oneof" inside of message "Entity".

// TimestampInfo is only intended to be read.  Attempts to update this
// entity have no effect, and should return an error status that the
// entity is read only.

message TimestampInfo {
  // The number of bits in the device's `timestamp_t` type.
  uint32 size_in_bits = 1;
  // The timestamp value of this device increments
  // `increments_per_period` times every `period_in_seconds` seconds.
  uint64 increments_per_period = 2;
  uint64 period_in_seconds = 3;
}

// The timestamp value can be read or written.  Note that if there are
// already timestamp values stored in tables or `Register` instances,
// they will not be updated as a result of writing this timestamp
// value.  Writing the device timestamp is intended only for
// initialization and testing.
```

[4] https://en.wikipedia.org/wiki/Precision_Time_Protocol

[5] https://en.wikipedia.org/wiki/Network_Time_Protocol

[1] The P4Runtime API, defined as a Google Protocol Buffer `.proto` file, can be found at https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

```
message Timestamp {
  bytes value = 1;
}
```

For every packet `P` that is processed by ingress and then egress, with the minimum possible latency in the packet buffer, it is guaranteed that the `egress_timestamp` value for that packet will be the same as, or slightly larger than, the `ingress_timestamp` value that the packet was assigned on ingress. By "slightly larger than", we mean that the difference `(egress_timestamp - ingress_timestamp)` should be a reasonably accurate estimate of this minimum possible latency through the packet buffer, perhaps truncated down to 0 if timestamps advance more slowly than this minimum latency.

Consider two packets such that at the same time (e.g. the same clock cycle), one is assigned its value of `ingress_timestamp` near the time it begins parsing, and the other is assigned its value of `egress_timestamp` near the time that it begins its egress processing. It is allowed that these timestamps differ by a few tens of nanoseconds (or by one "tick" of the timestamp, if one tick is larger than that time), due to practical difficulties in making them always equal.

Recall that the binary operators `+` and `-` on the `bit<W>` type in P4 are defined to perform wrap-around unsigned arithmetic. Thus even if a timestamp value wraps around from its maximum value back to 0, you can always calculate the number of ticks that have elapsed from timestamp $t1$ until timestamp $t2$ using the expression $(t2-t1)$ (if more than 2^W ticks have elapsed, there will be aliasing of the result). For example, if timestamps were $W >= 4$ bits in size, $t1 = 2^W - 5$, and $t2 = 3$, then $(t2 - t1) = 8$.

It is sometimes useful to minimize storage costs by discarding some bits of a timestamp value in a P4 program for use cases that do not need the full wrap time or precision. For example, an application that only needs to detect protocol timeouts with an accuracy of 1 second can discard the least significant bits of a timestamp that change more often than every 1 second.

Another example is an application that needed full precision of the least significant bits of a timestamp, but the combination of the control plane and P4 program are designed to examine all entries of a `Register` array where these partial timestamps are stored more often than once every 5 seconds, to prevent wrapping. In that case, the P4 program could discard the most significant bits of the timestamp so that the remaining bits wrap every 8 seconds, and store those partial timestamps in the `Register` instance.

4. Programmable blocks

The following declarations provide a template for the programmable blocks in the PSA. The P4 programmer is responsible for implementing controls that match these interfaces and instantiate them in a package definition.

The current implementation uses the same user-defined metadata structure for all the controls. An alternative design is to split the user-defined metadata into an input parameter and an output parameter for each block. The compiler will have to check that the out parameter of a block matches the in parameter of the subsequent block.

```
parser Parser<H, M>(packet_in buffer, out H parsed_hdr, inout M user_meta,
                    in psa_parser_input_metadata_t istd);

control VerifyChecksum<H, M>(in H hdr, inout M user_meta);

control Ingress<H, M>(inout H hdr, inout M user_meta,
                      PacketReplicationEngine pre,
                      in  psa_ingress_input_metadata_t  istd,
                      out psa_ingress_output_metadata_t ostd);
```

```
control Egress<H, M>(inout H hdr, inout M user_meta,
                     BufferingQueueingEngine bqe,
                     in  psa_egress_input_metadata_t  istd);

control ComputeChecksum<H, M>(inout H hdr, inout M user_meta);

control Deparser<H>(packet_out buffer, in H hdr);

package PSA_Switch<H, M>(Parser<H, M> p,
                         VerifyChecksum<H, M> vr,
                         Ingress<H, M> ig,
                         Egress<H, M> eg,
                         ComputeChecksum<H, M> ck,
                         Deparser<H> dep);
```